第5章

# 存储器系统

* 存储器系统的功能概览
* 存储器映射
* 存储器的各种访问属性
* 缺省的存储器访问许可
* 位带操作
* 非对齐数据传送
* 互斥访问
* 端模式

## 5.1 存储系统功能概览

CM3的存储器系统与从传统ARM架构的相比，已经有过脱胎换骨般的改革了：

1. 它的存储器映射是预定义的，并且还规定好了哪个位置使用哪条总线。
2. CM3的存储器系统支持所谓的“位带”（bit-band）操作。通过它，实现了对单一比特的原子操作。位带操作仅适用于一些特殊的存储器区域中，见本章论述。
3. CM3的存储器系统支持非对齐访问和互斥访问。这两个特性是直到了v7M时才出来的。

最后，CM3的存储器系统支持both小端配置和大端配置。

## 5.2 存储器映射

CM3只有一个单一固定的存储器映射。这一点极大地方便了软件在各种CM3单片机间的移植。举个简单的例子，各款CM3单片机的NVIC和MPU都在相同的位置布设寄存器，使得它们变得与具体器件无关。尽管如此，CM3定出的条条框框是粗线条的，它依然允许芯片制造商灵活细腻地分配存储器空间，以制造出各具特色的单片机产品。

存储空间的一些位置用于调试组件等私有外设，这个地址段被称为“私有外设区”。私有外设区的组件包括：

* + 闪存地址重载及断点单元(FPB)
  + 数据观察点单元(DWT)
  + 仪器化跟踪宏单元(ITM)
  + 嵌入式跟踪宏单元(ETM)
  + 跟踪端口接口单元(TPIU)
  + ROM表

在后续讨论调试特性的章节中，将详细讲述这些组件。

CM3的地址空间是4GB, 程序可以在代码区，内部SRAM区以及外部RAM区中执行。但是因为指令总线与数据总线是分开的，最理想的是把程序放到代码区，从而使取指和数据访问各自使用自己的总线，并行不悖。

让我们先看一看这4GB的粗线条划分：

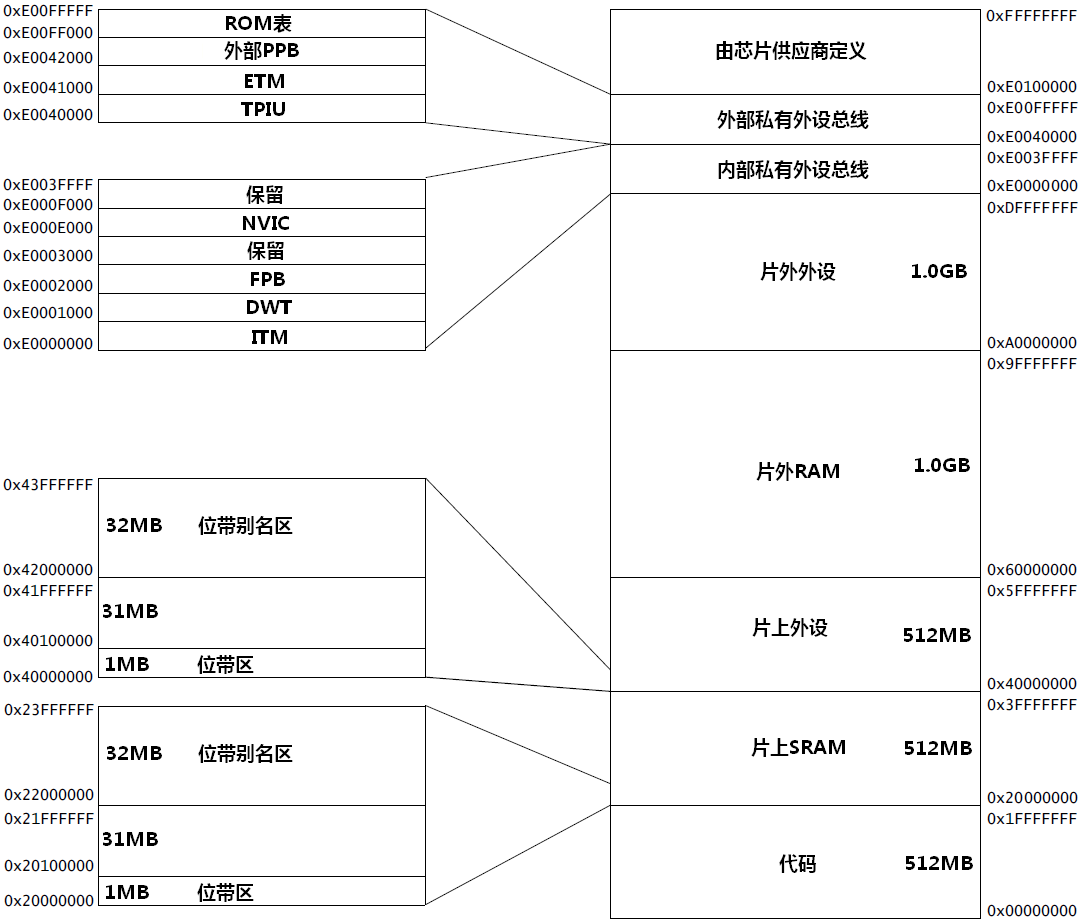


图5.1 Cortex-M3预定义的存储器映射

内部SRAM区的大小是512MB，用于让芯片制造商连接片上的SRAM，这个区通过系统总线来访问。在这个区的下部，有一个1MB的区间，被称为“位带区”。该位带区还有一个对应的、32MB的 “位带别名(alias)区”，容纳了8M个“位变量”（对比8051的只有128个位变量）。位带区对应的是最低的1MB地址范围，而位带别名区里面的每个字对应位带区的一个比特。位带操作只适用于数据访问，不适用于取指。通过位带的功能，可以把多个布尔型数据打包在单一的字中，却依然可以从位带别名区中，像访问普通内存一样地使用它们。位带别名区中的访问操作是原子的，消灭了传统的“读－改－写”三步曲。位带操作的细节待会还要讲到。

地址空间的另一个512MB范围由片上外设（的寄存器）使用。这个区中也有一条32MB的位带别名，以便于快捷地访问外设寄存器，用法与内部SRAM区中的位带相同。例如，可以方便地访问各种控制位和状态位。要注意的是，外设区内不允许执行指令。

还有两个1GB的范围，分别用于连接外部RAM和外部设备，它们之中没有位带。两者的区别在于外部RAM区允许执行指令，而外部设备区则不允许。

最后还剩下0.5GB的隐秘地带，CM3内核的闺房就在这里面，包括了系统级组件，内部私有外设总线s，外部私有外设总线s，以及由提供者定义的系统外设。

私有外设总线有两条：

* AHB私有外设总线，只用于CM3内部的AHB外设，它们是：NVIC, FPB, DWT和ITM。
* APB私有外设总线，既用于CM3内部的APB设备，也用于外部设备（这里的“外部”是对内核而言）。CM3允许器件制造商再添加一些片上APB外设到APB私有总线上，它们通过APB接口来访问。

NVIC所处的区域叫做“系统控制空间（SCS）”，在SCS里的除了NVIC外，还有SysTick、MPU以及代码调试控制所用的寄存器，如图5.2所示：

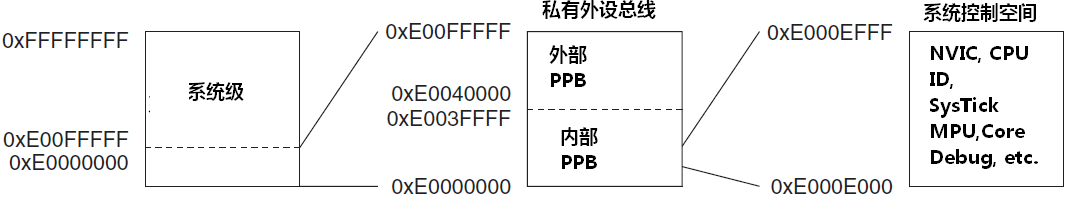


图5.2 系统控制空间(SCS)

最后，未用的提供商指定区也通过系统总线来访问，但是不允许在其中执行指令。

CM3中的MPU是选配的，由芯片制造商决定是否配上。

上述的存储器映射只是个粗线条的模板，半导体厂家会提供更展开的图示，来表明芯片中片上外设的具体分布，RAM与ROM的容量和位置信息。

## 5.3 存储器的各种访问属性

CM3在定义了存储器映射之外，还为存储器的访问规定了4种属性，分别是：

* + 可否缓冲(Bufferable)
  + 可否缓存(Cacheable)
  + 可否执行(Executable)
  + 可否共享(Sharable)

如果配了MPU，则可以通过它配置不同的存储区，并且覆盖缺省的访问属性。CM3片内没有配备缓存，也没有缓存控制器，但是允许在外部添加缓存。通常，如果提供了外部内存，芯片制造商还要附加一个内存控制器，它可以根据可否缓存的设置，来管理对片内和片外RAM的访问操作。地址空间可以通过另一种方式分为8个512MB等份：

1. 代码区（0x0000\_0000- 0x1FFF\_FFFF）。该区是可以执行指令的，缓存属性为WT（“写通”，Write Through），即不可以缓存。此区亦允许布设数据存储器。在此区上的数据操作是通过数据总线接口的（估计读数据使用D-Code，写数据使用System），且在此区上的写操作是缓冲的。
2. SRAM区（0x2000\_0000 – 0x3FFF\_FFFF）。此区用于片内SRAM，写操作是缓冲的，并且可以选择WB-WA(Write Back, Write Allocated)缓存属性。此区亦可以执行指令，以允许把代码拷贝到内存中执行——常用于固件升级等维护工作。
3. 片上外设区(0x4000\_0000 – 0x5FFF\_FFFF)。该区用于片上外设，因此是不可缓存的，也不可以在此区执行指令（这也称为eXecute Never, XN。ARM的参考手册大量使用此术语）。
4. 外部RAM区的前半段（0x6000\_0000 - 0x7FFF\_FFFF）。该区可用于布设片上RAM或片外RAM，可缓存（缓存属性为WB-WA），并且可以执行指令。
5. 外部RAM区的后半段（0x8000\_0000 – 0x9FFF\_FFFF）。除了不可缓存(WT)外，同前半段。
6. 外部外设区的前半段(0xA000\_0000 – 0xBFFF\_FFFF)。用于片外外设的寄存器，也用于多核系统中的共享内存（需要严格按顺序操作，即不可缓冲）。该区也是个不可执行区。
7. 外部外设区的后半段(0xC000\_0000 – 0xDFFF\_FFFF)。目前与前半段的功能完全一致。
8. 系统区(0xE000\_0000 – 0xFFFF\_FFFF)。此区是私有外设和供应商指定功能区。此区不可执行代码。系统区涉及到很多关键部位，因此访问都是严格序列化的（不可缓存，不可缓冲）。而供应商指定功能区则是可以缓存和缓冲的。

需要注意的是，在CM3的第一版中，代码区的存储器属性是被硬件连接成可缓存可缓冲的，无法通过MPU来更改。

译者添加

写通，写回，与写时申请

* 写回(Write Back)： 写入的数据先逗留在缓存中，待到必要时再落实到最终目的地，这也是cache的最基本职能，用于改善数据传送的效率，减少对访问主存储器的访问操作。
* 写通(Write Through)： 写操作“穿透”中途的缓存，直接落入最终的存储器目的地址中。可见，写通操作架空了cache，但它使写操作的结果立即生效。这常用于和片上外设或其它处理器共享的内存中，如显卡的显存，片上外设寄存器，以及双核系统中的共享内存。写通操作和C中的“volatile”可以配合使用——带volatile属性的变量往往放到写通型地址区间中。
* 写时申请(Write Allocate)：俺也不太清楚~

## 5.4 存储器的缺省访问许可

CM3有一个缺省的存储访问许可，它能防止使用户代码访问系统控制存储空间，保护NVIC、MPU等关键部件。缺省访问许可在下列条件时生效：

* + 没有配备MPU
  + 配备了MPU，但是MPU被除能

如果启用了MPU，则MPU可以在地址空间中划出若干个regions，并为不同的region规定不同的访问许可权限。

缺省的存储器访问许可权限如表5.1所示

表5.1 存储器的缺省访问许可

|  |  |  |
| --- | --- | --- |
| 存储器区域 | 地址范围 | 用户级许可权限 |
| 代码区 | 0000\_0000 – 1FFF\_FFFF | 无限制 |
| 片内SRAM | 2000\_0000 – 3FFF\_FFFF | 无限制 |
| 片上外设 | 4000\_0000 – 5FFF\_FFFF | 无限制 |
| 外部RAM | 6000\_0000 – 9FFF\_FFFF | 无限制 |
| 外部外设 | A000\_0000 – DFFF\_FFFF | 无限制 |
| ITM | E000\_0000 – E000\_0FFF | 可以读。对于写操作，除了用户级下允许时的stimulus端口外，全部忽略 |
| DWT | E000\_1000 – E000\_1FFF | 阻止访问，访问会引发一个总线fault |
| FPB | E000\_2000 – E000\_3FFF | 阻止访问，访问会引发一个总线fault |
| NVIC | E000\_E000 – E000\_EFFF | 阻止访问，访问会引发一个总线fault。但有个例外：软件触发中断寄存器可以被编程为允许用户级访问。 |
| 内部PPB | E000\_F000 – E003\_FFFF | 阻止访问，访问会引发一个总线fault |
| TPIU | E004\_0000 – E004\_0FFF | 阻止访问，访问会引发一个总线fault |
| ETM | E004\_1000 – E004\_1FFF | 阻止访问，访问会引发一个总线fault |
| 外部PPB | E004\_2000 – E004\_2FFF | 阻止访问，访问会引发一个总线fault |
| ROM表 | E00F\_F000 – E00F\_FFFF | 阻止访问，访问会引发一个总线fault |
| 供应商指定 | E010\_0000 – FFFF\_FFFF | 无限制 |

当一个用户级访问被阻止时，会立即产生一个总线fault。

## 5.5 位带操作

支持了位带操作后，可以使用普通的加载/存储指令来对单一的比特进行读写。在CM3中，有两个区中实现了位带。其中一个是SRAM区的最低1MB范围，第二个则是片内外设区的最低1MB范围。这两个位带中的地址除了可以像普通的RAM一样使用外，它们还都有自己的“位带别名区”，位带别名区把每个比特膨胀成一个32位的字。当你通过位带别名区访问这些字时，就可以达到访问原始比特的目的。

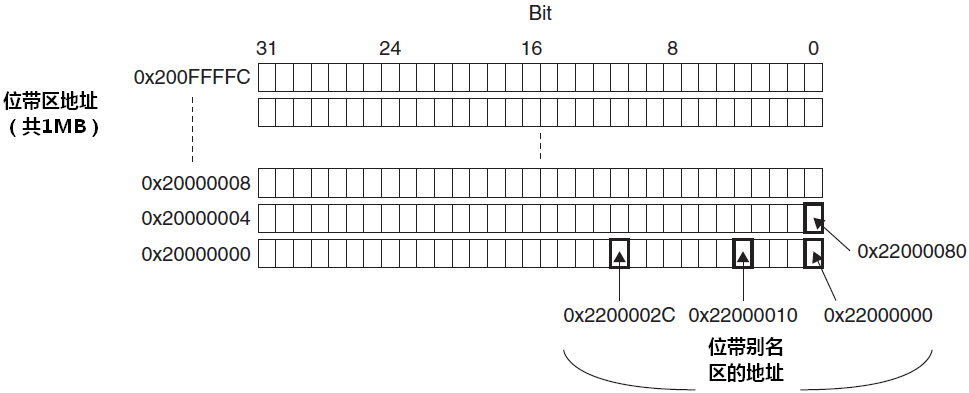


图5.3A 位带区与位带别名区的膨胀关系图A：

译者添加 下图从另一个侧面演示比特的膨胀对应关系

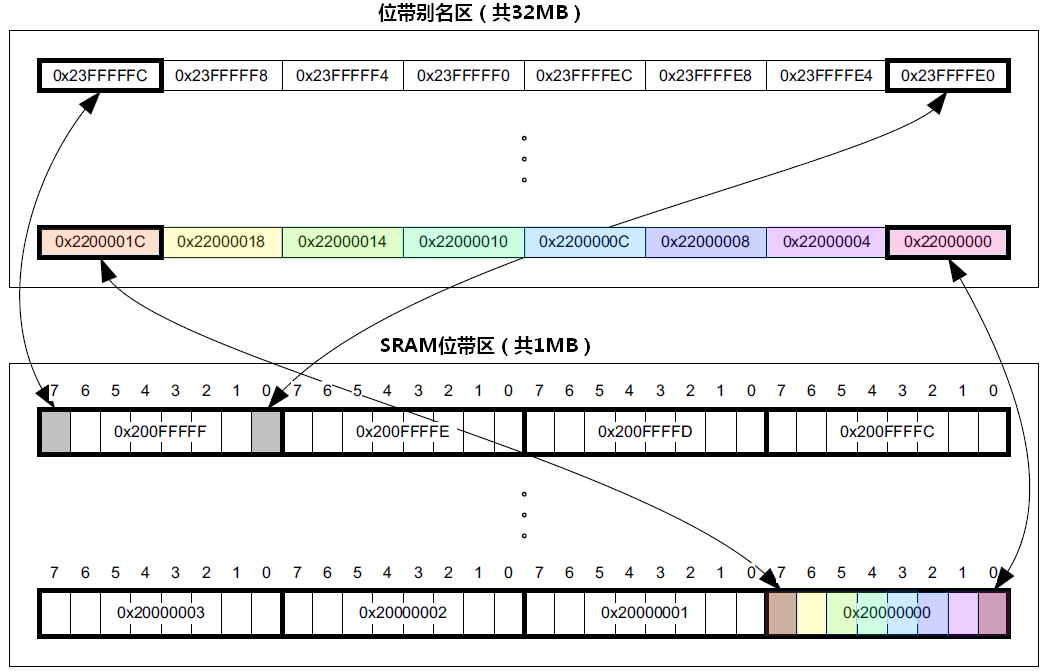


图5.3B 位带区与位带别名区的膨胀对应关系图B

举例：欲设置地址0x2000\_0000中的比特2，则使用位带操作的设置过程如下图所示：

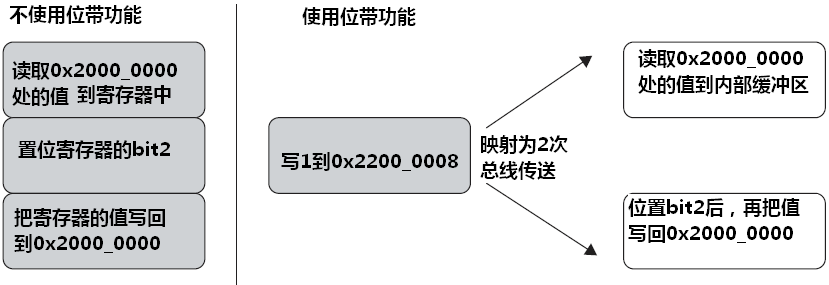


图5.4 写数据到位带别名区

对应的汇编代码如图5.5所示

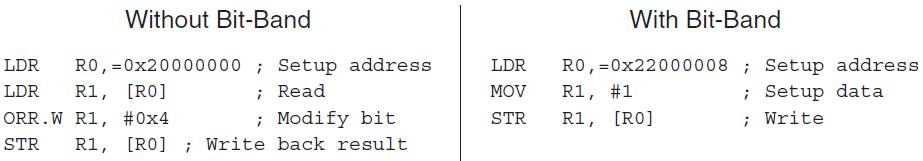


图5.5 位带操作与普通操作的对比，在汇编程序的角度上

位带读操作相对简单些：

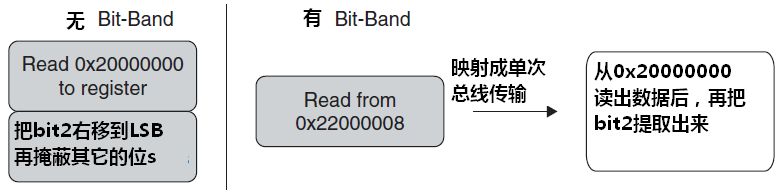


图5.6 从位带别名区中读取比特

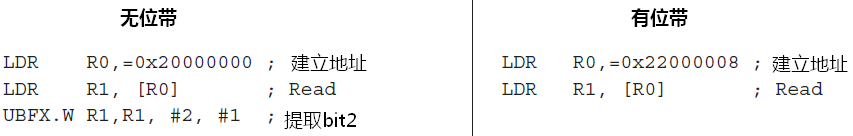


图5.7 读取比特时传统方法与位带方法的比较

位带操作的概念其实30年前就有了，那还是8051单片机开创的先河。如今，CM3将此能力进化，这里的位带操作是8051位寻址区的威力极度加强版。

CM3使用如下术语来表示位带存储的相关地址

* + 位带区： 支持位带操作的地址区
  + 位带别名： 对别名地址的访问最终会变换成对位带区的访问（注意：这中途有一个地址映射过程）

在位带区中，每个比特都映射到别名地址区的一个字——这是个只有LSB才有效的字。当一个别名地址被访问时，会先把该地址变换成位带地址。对于读操作，读取位带地址中的一个字，再把需要的位右移到LSB，并把LSB返回。对于写操作，把需要写的位左移至对应的位序号处，然后执行一个原子的“读－改－写”过程。

支持位带操作的两个内存区的范围是：

0x2000\_0000-0x200F\_FFFF（SRAM区中的最低1MB）

0x4000\_0000-0x400F\_FFFF（片上外设区中的最低1MB）

对于SRAM位带区的某个比特，记它所在字节地址为A,位序号为n(0<=n<=7)，则该比特在别名区的地址为：

AliasAddr＝ 0x22000000+((A-0x20000000)\*8+n)\*4 =0x22000000+ (A-0x20000000)\*32 + n\*4

对于片上外设位带区的某个比特，记它所在字节的地址为A,位序号为n(0<=n<=7)，则该比特在别名区的地址为：

AliasAddr＝ 0x42000000+((A-0x40000000)\*8+n)\*4 =0x42000000+ (A-0x40000000)\*32 + n\*4

上式中，“\*4”表示一个字为4个字节，“\*8”表示一个字节中有8个比特。

对于SRAM内存区，位带别名的重映射如表5.2所示：

表5.2 SRAM区中的位带地址映射

|  |  |
| --- | --- |
| 位带区 | 等效的别名地址 |
| 0x20000000.0 | 0x22000000.0 |
| 0x20000000.1 | 0x22000004.0 |
| 0x20000000.2 | 0x22000008.0 |
| … |  |
| 0x20000000.31 | 0x2200007C.0 |
| 0x20000004.0 | 0x22000080.0 |
| 0x20000004.1 | 0x22000084.0 |
| 0x20000004.2 | 0x22000088.0 |
| … |  |
| 0x200FFFFC.31 | 0x23FFFFFC.0 |

对于片上外设，映射关系如下表所示：

表5.3 片上外设区中的位带地址映射

|  |  |
| --- | --- |
| 位带区 | 等效的别名地址 |
| 0x40000000.0 | 0x42000000.0 |
| 0x40000000.1 | 0x42000004.0 |
| 0x40000000.2 | 0x42000008.0 |
| … |  |
| 0x40000000.31 | 0x4200007C.0 |
| 0x40000004.0 | 0x42000080.0 |
| 0x40000004.1 | 0x42000084.0 |
| 0x40000004.2 | 0x42000088.0 |
| … |  |
| 0x400FFFFC.31 | 0x43FFFFFC.0 |

这里再不嫌啰嗦地举一个例子：

1. 在地址0x20000000处写入0x3355AACC
2. 读取地址0x22000008。本次读访问将读取0x20000000，并提取比特2，值为1。
3. 往地址0x22000008处写0。本次操作将被映射成对地址0x20000000的“读－改－写”操作（原子的），把比特2清0。
4. 现在再读取0x20000000，将返回0x3355AAC8（bit[2]已清零）。

位带别名区的字只有LSB有意义。另外，在访问位带别名区时，不管使用哪一种长度的数据传送指令（字/半字/字节），都把地址对齐到字的边界上，否则会产生不可预料的结果。

### 5.5.1 位带操作的优越性

位带操作有什么优越性呢？最容易想到的就是通过GPIO的管脚来单独控制每盏LED的点亮与熄灭。另一方面，也对操作串行接口器件提供了很大的方便（典型如74HC165,CD4094）。总之位带操作对于硬件I/O密集型的底层程序最有用处了。对于大范围使用位标志的系统程序来说，位带机制也是一大福音。

CM3中还有一个称为“bit-bang”的概念，它通常是通过“bit-band”实现的，但是它俩在学术上是两个不同的概念（不过本书中除了这里之外，就再也没有提到过bit-bang——译注）。

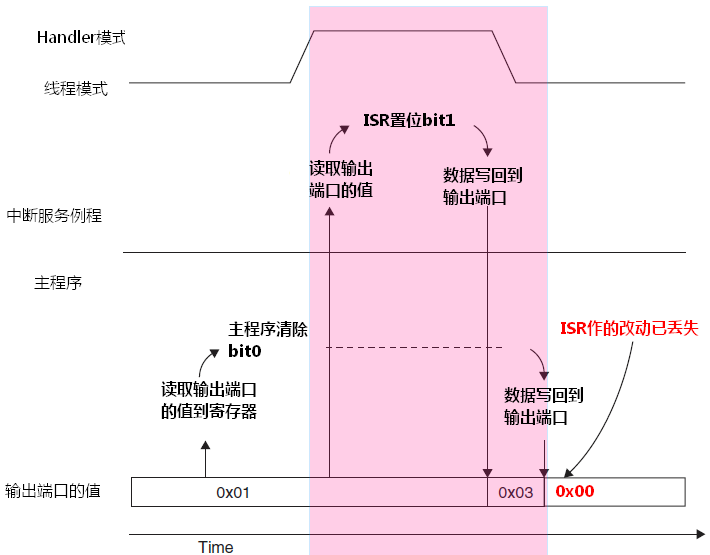
位带操作还能用来化简跳转的判断。当跳转依据是某个位时，以前必须这样做：

* + 读取整个寄存器
  + 掩蔽不需要的位
  + 比较并跳转

现在只需：

* + - 从位带别名区读取状态位
    - 比较并跳转

使代码更简洁，这只是位带操作优越性的初等体现，位带操作还有一个重要的好处是在多任务中，用于实现共享资源在任务间的“互锁”访问。多任务的共享资源必须满足一次只有一个任务访问它——亦即所谓的“原子操作”。以前的读－改－写需要3条指令，导致这中间留有两个能被中断的空当。于是可能会出现如下图所示的紊乱危象：



**ISR所作的改动已丢失！**

图5.8 共享资源在紊乱危象下丢失数据演示

同样的紊乱危象可以出现在多任务的执行环境中。其实，图5.8所演示的情况可以看作是多任务的一个特例：主程序是一个任务，ISR是另一个任务，这两个任务并发执行。

通过使用CM3的位带操作，就可以消灭上例中的紊乱危象。CM3把这个“读－改－写”做成一个硬件级别支持的原子操作，不能被中断，如图5.9所演示

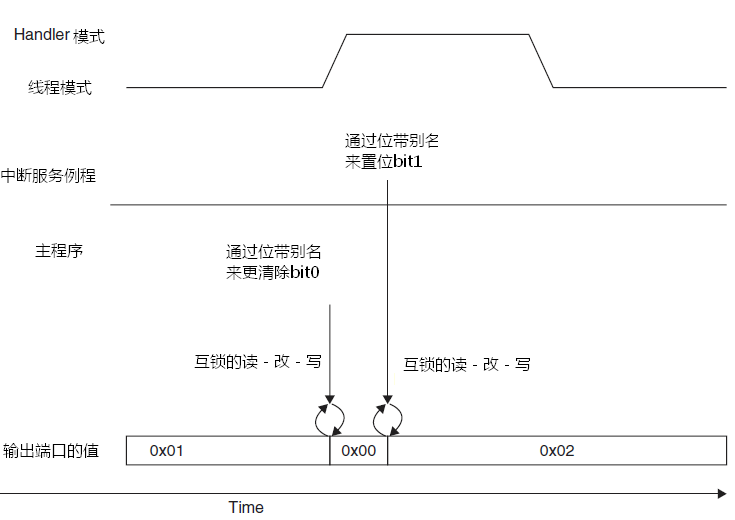


图5.9 通过位带操作实现互锁访问，从而避免紊乱危象的演示

同样道理，多任务环境中的紊乱危象亦可以通过互锁访问来避免。

### 5.5.2 其它数据长度上的位带操作

位带操作并不只限于以字为单位的传送。亦可以按半字和字节为单位传送。例如，可以使用LDRB/STRB来以字节为长度单位去访问位带别名区，同理可用于LDRH/STRH。但是不管用哪一个对子，都必须保证目标地址对齐到字的边界上。

### 5.5.3 在C语言中使用位带操作

不幸的是，在C编译器中并没有直接支持位带操作。比如，C编译器并不知道同一块内存能够使用不同的地址来访问，也不知道对位带别名区的访问只对LSB有效。欲在C中使用位带操作，最简单的做法就是#define一个位带别名区的地址。例如：

#define DEVICE\_REG0 ((volatile unsigned long \*) (0x40000000))

#define DEVICE\_REG0\_BIT0 ((volatile unsigned long \*) (0x42000000))

#define DEVICE\_REG0\_BIT1 ((volatile unsigned long \*) (0x42000004))

...

\*DEVICE\_REG0 = 0xAB; //使用正常地址访问寄存器

...

\*DEVICE\_REG0 = \*DEVICE\_REG0 | 0x2; //使用传统方法设置bit1

...

\*DEVICE\_REG0\_BIT1 = 0x1; // 通过位带别名地址设置bit1

为简化位带操作，也可以定义一些宏。比如，我们可以建立一个把“位带地址＋位序号”转换成别名地址的宏，再建立一个把别名地址转换成指针类型的宏：

//把“位带地址＋位序号”转换成别名地址的宏

#define BITBAND(addr, bitnum) ((addr & 0xF0000000)+0x2000000+((addr & 0xFFFFF)<<5)+(bitnum<<2))

//把该地址转换成一个指针

#define MEM\_ADDR(addr) \*((volatile unsigned long \*) (adr))

在此基础上，我们就可以如下改写代码：

MEM\_ADDR(DEVICE\_REG0) = 0xAB; //使用正常地址访问寄存器

MEM\_ADDR(DEVICE\_REG0)= MEM\_ADDR(DEVICE\_REG0) | 0x2; //传统做法

MEM\_ADDR(BITBAND(DEVICE\_REG0,1)) = 0x1; //使用位带别名地址

请注意：当使用位带功能时，要访问的变量必须用volatile来定义。因为C编译器并不知道同一个比特可以有两个地址。所以就要通过volatile，使得编译器每次都如实地把新数值写入存储器，而不再会出于优化的考虑，在中途使用寄存器来操作数据的复本，直到最后才把复本写回——这会导致按不同的方式访问同一个位会得到不一致的结果（可能被优化到不同的寄存器来保存中间结果——译注）

译者添加

在GCC和RealView MDK (即Keil) 开发工具中，允许定义变量时手工指定其地址。如：

volatile unsigned long bbVarAry[7] \_\_attribute\_\_(( at(0x20003014) ));

volatile unsigned long\* const pbbaVar= (void\*)(0x22000000+0x3014\*8\*4);

这样，就在0x20003014处分配了7个字，共得到了32\*7=224个比特。

在long\*后面的“const”通知编译器：该指针不能再被修改而指向其它地址。

注意：at()中的地址必须对齐到4字节边界。

再使用这些比特时，可以通过如下的形式：

pbbaVar[136]=1; //置位第136号比特

不过这有个局限：编译器无法检查是否下标越界。那为什么不定义成“bbaVarAry[224]”的数组呢？这也是一个编译器的局限：它不知道这个数组其实就是bbVarAry[7]，从而在计算程序对内存的占用量上，会平白无故地多计入224\*4个字节。对于指针形式的定义，可以使用宏定义，为每个需要使用的比特取一个字面值的名字，在下标中只使用字面值名字，不再写真实的数字，就可以极大程度地避免数组越界。

请注意：在定义这“两个”变量时，前面加上了“volatile”。如果不再使用bbVarAry来访问这些比特，而仅仅使用位带别名的形式访问时，这两个volatile均不再需要。

## 5.6 非对齐数据传送

CM3支持在单一的访问中使用非（地址）对齐的传送，数据存储器的访问无需对齐。在以前，ARM处理器只允许对齐的数据传送。这种对齐是说：以字为单位的传送，其地址的最低两位必须是0；以半字为单位的传送，其地址的LSB必须是0；以字节为单位的传送则无所谓对不对齐。如果使用0x1001,0x1002或0x1003这样的地址做字传送，在以前的ARM处理器中则会触发一个数据流产（Data abort）异常——与CM3中总线fault异常的作用相同。

那么，非对齐访问看起来是什么样子呢？图5.12-5.16给出了5个例子。对于字的传送来说，任何一个不能被4整除的地址都是非对齐的。而对于半字，任何不能被2整除的地址（也就是奇数地址）都是非对齐的：

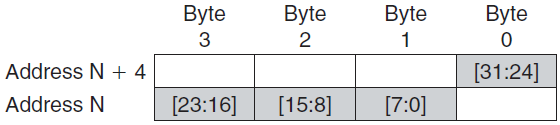


图5.12 非对齐传送示例1

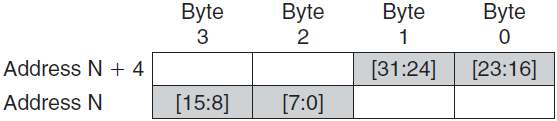


图5.13 非对齐传送示例2

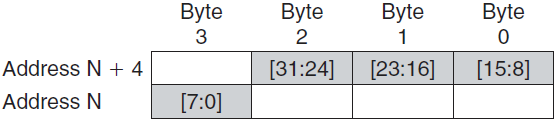


图5.14 非对齐传送示例3

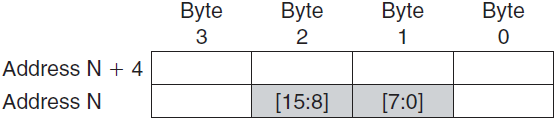


图5.15 非对齐传送示例4

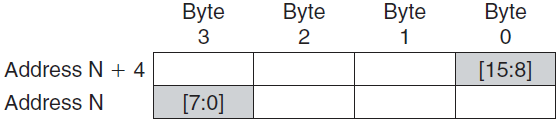


图5.16 非对齐传送示例5

在CM3中，非对齐的数据传送只发生在常规的数据传送指令中，如LDR/LDRH/LDRSH。其它指令则不支持，包括：

* + 多个数据的加载/存储(LDM/STM)
  + 堆栈操作PUSH/POP
  + 互斥访问(LDREX/STREX)。如果非对齐会导致一个用法fault
  + 位带操作。因为只有LSB有效，非对齐的访问会导致不可预料的结果。

事实上，在内部是把非对齐的访问转换成若干个对齐的访问的，这种转换动作由处理器总线单元来完成。这个转换过程对程序员是透明的，因此写程序时不必操心。但是，因为它通过若干个对齐的访问来实现一个非对齐的访问，会需要更多的总线周期。事实上，节省内存有很多方法，但没有一个是通过压缩数据的地址，不惜破坏对齐性的这种旁门左道。因此，应养成好习惯，总是保证地址对齐，这也是让程序可以移植到其它ARM芯片上的必要条件。

为此，可以编程NVIC，使之监督地址对齐。当发现非对齐访问时触发一个fault。具体的办法是设置“配置控制寄存器”中的UNALIGN\_TRP位。这样，在整个调试期间就可以保证非对齐访问能当场被发现。

## 5.7 互斥访问

细心的读者可能会发现，CM3中没有类似“SWP”的指令。在传统的ARM处理器中，SWP指令是实现互斥体所必需的。到了CM3，由所谓的互斥访问取代了SWP指令，以实现更加老练的共享资源访问保护机制。

互斥体在多任务环境中使用，也在中断服务例程和主程序之间使用，用于给任务申请共享资源（如一块共享内存）。在某个（排他型）共享资源被一个任务拥有后，直到这个任务释放它之前，其它任务是不得再访问它的。为建立一个互斥体，需要定义一个标志变量，用来指示其对应的共享资源是否已经被某任务拥有。当另一个任务欲取得此共享资源时，它要先检查这个互斥体，以获知共享资源是否无人使用。在传统的ARM处理器中，这种检查操作是通过SWP指令来实现的。SWP保证互斥体检查是原子操作的，从而避免了一个共享资源同时被两个任务占有（这是紊乱危象的一种常见表现形式）。

在新版的ARM处理器中，读/写访问往往使用不同的总线，导致SWP无法再保证操作的原子性，因为只有在同一条总线上的读/写能实现一个互锁的传送。因此，互锁传送必须用另外的机制实现，这就引入了“互斥访问”。互斥访问的理念同SWP非常相似，不同点在于：在互斥访问操作下，允许互斥体所在的地址被其它总线master访问，也允许被其它运行在本机上的任务访问，但是CM3能够“驳回”有可能导致竞态条件的互斥写操作。

互斥访问分为加载和存储，相应的指令对子为LDREX/STREX, LDREXH/STREXH, LDREXB/STREXB，分别对应于字/半字/字节。为了介绍方便，以LDREX/STREX为例讲述它们的使用方式。

LDREX/STREX的语法格式为：

LDREX Rxf, [Rn, #offset]

STREX Rd, Rxf, [Rn, #offset]

（本节的以下内容是译者改编的）

LDREX的语法同LDR相同，这里不再赘述。而STREX则不同。STREX指令的执行是可以被“驳回”的。当处理器同意执行STREX时，Rxf的值被存储到(Rn+offset)处，并且把Rd的值更新为0。但若处理器驳回了STREX的执行，则不会发生存储动作，并且把Rd的值更新为1。

其实，奥妙就在于这个“驳回”的规则上。规则可宽可严，最严格的规则是：

当遇到STREX指令时，仅当在它之前执行过LDREX指令，且在最近的一条LDREX指令执行后，没有执行过其它的STR/STREX指令，才允许执行本条STREX指令——也就是说只有在LDREX执行后，从时间上与之距离最近的一条STREX才能成功执行。

其它情况下，驳回此STREX。包括：

* + 中途有其它的STR指令执行
  + 中途有其它的STREX指令执行。

这种最严格的规则也是最容易实现的规则。在CM3的技术参考手册中，推荐实现者标记出一段有限的地址，只在这段地址中适用互斥访问的规则，而不要对所有4GB都限制住。这段地址通常是从LDREX指令族给出的地址开始，长度在16字节至4K字节范围内。但芯片制造商可能更倾向严格的规则。

在使用互斥访问时，LDREX/STREX必须成对使用。

为什么这种有条件的驳回可以避免紊乱危象呢？让我们举个简单的例子来演示。这个例子由主程序和一个中断服务例程组成。主程序尝试对(R0)自增两次，中断服务例程则把(R0).5置位。计(R0)的初始值为0。

MainProgram

;第一次互斥自增

TryInc1st

LDREX r2, [R0]

ADD r2, #1

;执行到这里时，处理器接收到外中断3请求，于是转到其中断服务例程ISREx3中

STREX R1, R2, [R0] ; STREX被驳回，R1=1，(R0)=0x20

TryInc2nd

;第二次互斥自增

LDREX r2, [R0]

ADD r2, #1

STREX R1, R2, [R0] ; STREX得到执行，R1=0，(R0)=0x21

…

ISREx3

;处理器已经自动把R0-R3, R12, LR, PC, PSR压入栈

LDR R2, [R0]

ORR R2, #0x20

STR R2, [R0] ;在ISREx3中设置了(r0)的Bit2

BX LR ;返回时，处理器会自动把R0-R3,R12,LR,PC,PSR弹出堆栈

上例中，主程序在即将执行第一条STREX时，产生了外部中断#3。处理器打断主程序的执行，进入其服务例程ISREx3，它对(R0)执行了一个写操作(STR)，因此在ISRExt3返回后，STREX不再是LDREX执行后的第一条存储指令，故而被驳回。从而ISREx3对(R0)的改动就不会遭到破坏。随后主程序再次尝试自增运算，这一次在STREX执行前没有其它任何形式的存储指令，所有STREX成功执行。

如果主程序使用普通的STR会怎么样呢？对于第一次自增，主程序的R2=1，于是执行后(R0)=1，结果，中断服务程序对(R0)的改动在此丢失！

上例是为演示方便才写了第2次自增尝试。实际情况是用循环实现的：

TryInc

LDREX r2, [R0]

ADD r2, #1

STREX R1, R2, [R0]

CMP R1, #1 ;检查STREX是否被驳回

BEQ TryInc ;如果发现STREX被驳回，则重试。

LDREX/STREX的工作原理其实很简单。仍然以上一段程序为例：当执行了LDREX后，处理器会在内部标记出一段地址。原则上，这段地址从R0开始，范围由芯片制造商定义。技术手册推荐的范围是在4字节至4KB之间，但是很多粗线条的实现会标记整个4GB的地址。在标记以后，对于第一个执行到的STR/STREX指令，只要其存储的地址落在标记范围内，就会清除此标记（对于整个4GB地址都被标记的情况，则任何存储指令都会清除此标记）。如果先后执行了两次LDREX，则以后一个LDREX标记的地址为准。

执行STREX时，会先检查有没有做出过标记，如果有，还要检查存储地址是否落在标记范围内。只有通过了这两个关卡，STREX才会执行。否则，就驳回STREX。

当使用互斥访问时，在CM3总线接口上的内部写缓冲会被旁路，即使是MPU规定此区是可以缓冲的也不行。这保证了互斥体的更新总能在第一时间内完成，从而保证数据在各个总线主机(master)之间是一致的。So系统的设计师如果设计多核系统，则必须保证各核之间看到的数据也是一致的。

译者添加的选读材料——互斥访问的深入研究

互斥访问可以递归使用，且最后一次递归的LDREX/STREX对子最先完成。如下例所示：

LDREXTestRecursive

ldr r3, =N ;递归次数N，是一个预定义的常数

LoopWrapper

push {r0-r2, lr}

ldr r0, =0x20003000

sub r3, #1

TryInc

ldrex r1, [r0]

add r1, #1

ldr lr, =DoSTREXRcsv

cmp r3, #0

bne LoopWrapper

DoSTREXRcsv

strex r2, r1, [r0]

cmp r2, #1

beq TryInc

pop {r0-r2, pc}

若执行前(0x20003000)=0，则执行后(0x20003000)＝N，且函数被递归调用N次。这段代码的工作流程难以用文字说清，一定要用模拟器跑过才容易理解

本例只是为了抛砖引玉。在实际的程序中，极少会这样钻牛角尖地直接递归。但是在多任务环境下，底层的函数库往往会“重入”，这也和递归的情形很相似。另外，当读者在本书后面看到“自旋锁”的解释时，说不定也会回想起这里呢！

## 5.8 端模式

CM3支持both小端模式和大端模式。但是，单片机其它部分的设计，包括总线的连接，内存控制器以及外设的性质等，也共同决定可以支持的内存类型。所以在设计软件之前，一定要先在单片机的数据手册上查清楚可以使用的端。在绝大多数情况下，基于CM3的单片机都使用小端模式——为了避免不必要的麻烦，在这里推荐读者清一色地使用小端模式。

CM3中对大端模式的定义还与ARM7的不同（小端的定义都是相同的）。在ARM7中，大端的方式被称为“字不变大端”，而在CM3中，使用的是“字节不变大端”。如表5.4所示 。

表5.4 CM3的字节不变大端：存储器视图

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 地址，长度 | Bits 31-24 | Bits 23-16 | Bits 15-8 | Bits 7-0 |
| 0x1000，字 | D[7:0] | D[15:8] | D[23:16] | D[31:24] |
| 0x1000，半字 | D[7:0] | D[15:8] | - | - |
| 0x1002，半字 |  |  | D[7:0] | D[15:8] |
| 0x1000，字节 | D[7:0] |  |  |  |
| 0x1001，字节 |  | D[7:0] |  |  |
| 0x1002，字节 |  |  | D[7:0] |  |
| 0x1003，字节 |  |  |  | D[7:0] |

表5.5 CM3的字节不变大端：在AHB上的数据

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 地址，长度 | Bits 31-24 | Bits 23-16 | Bits 15-8 | Bits 7-0 |
| 0x1000，字 | D[7:0] | D[15:8] | D[23:16] | D[31:24] |
| 0x1000，半字 |  |  | D[7:0] | D[15:8] |
| 0x1002，半字 | D[7:0] | D[15:8] |  |  |
| 0x1000，字节 |  |  |  | D[7:0] |
| 0x1001，字节 |  |  | D[7:0] |  |
| 0x1002，字节 |  | D[7:0] |  |  |
| 0x1003，字节 | D[7:0] |  |  |  |

请注意：在AHB总线上的BE-8模式下，数据字节lane的传送格式是与小端模式一致的。

这是不同于ARM7TDMI的行为，它在大端模式下会有另一种总线车道（lane）安排，如表5.6所示。

表5.6 ARM7的字不变大端：在AHB上的数据

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 地址，长度 | Bits 31-24 | Bits 23-16 | Bits 15-8 | Bits 7-0 |
| 0x1000，字 | D[7:0] | D[15:8] | D[23:16] | D[31:24] |
| 0x1000，半字 | D[7:0] | D[15:8] | - | - |
| 0x1002，半字 |  |  | D[7:0] | D[15:8] |
| 0x1000，字节 | D[7:0] |  |  |  |
| 0x1001，字节 |  | D[7:0] |  |  |
| 0x1002，字节 |  |  | D[7:0] |  |
| 0x1003，字节 |  |  |  | D[7:0] |

在CM3中，是在复位时确定使用哪种端模式的，且运行时不得更改。指令预取永远使用小端模式，在配置控制存储空间的访问也永远使用小端模式（包括NVIC, FPB等）。另外，外部私有总线地址区0xE0000000至0xE00FFFFF也永远使用小端模式。

当你的So设计不支持大端模式，却有一些外设包含了大端模式时，可以轻易地使用REV/REVH指令来完成端模式的转换。